# Writing Stack Based Overflows on Windows
# Part I – Basic Concepts

Nish Bhalla, 31st May, 2005

(Nish [a-t] SecurityCompass.com)

www.SecurityCompass.com

# Introduction

This series focuses on writing stack based overflows for windows. Before we get into writing exploits, it is important to get some basic concepts cleared up.

The first of this multipart article will focus on explaining the basic concepts of how a executable is loaded into memory and executed. The second part will focus on basic assembly instructions, the third part will delve into what stack overflows are, why they exists and how to write exploits for vulnerable applications for a local exploit and finally the part four will show how to write your own shellcode and how to exploit a remote buffer overflow.

This article provides detailed examples, which can be used to learn the concepts. The examples are written and based on Windows XP (No SP). The examples have been tested on Visual Studio 6.0 and Visual Studio 7.0. Some of the examples are also demonstrated in Visual Studio .NET (or VS 7.0). These examples are shown to display the new security features implemented by Microsoft. The Visual Studio .NET development environment can be used to write the applications and has some built-in default security measures to help prevent some of the attacks displayed here. For this series of examples, the "/GS" flag option must be disabled while compiling all the examples to ensure the exploits work and the applications are still vulnerable. As the article goes on different tools and utilities are introduced and links to where they can be downloaded from are also provided. The examples are available for download from the site as well.

One of the questions that might come to mind before you read this article is, why another article on writing exploits? Before I learnt how to write exploits I attempted to read some excellent articles on writing exploits, but they always made some assumptions on pre-requisite knowledge, I have attempted to cover a lot of that in this series. Secondly, most of the articles used sample applications to demonstrate their examples however; the applications that they choose to demonstrate those examples with, were obsolete over a period of time or were no longer available for download. I often found spending more time searching for the exact version of the application.

Many developers are under the false impression that the usage of the "GS" flag prevents attacks against all buffer overflow exploits. Microsoft themselves also state that it doesn't prevent against all types of stack based attacks (http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp). Additionally, there are many other development environments that are used which do not have this feature implemented. Applications developed in those development environments are still vulnerable to these exploit technique.

The articles were written after references numerous links and documents, the important ones of which have been provided in the "Reference for further reading section".

Please do not hesitate to contact me if you have any questions or suggestions (articles[a-t]securitycompass.com).

Lastly, I would like to thank Jamie, Nitesh, Dhillon for critiquing the doc.

## Basic Concepts: Part I / IV

Every application is assigned 4 GB of *virtual* memory space (even though the *physical* memory might be much lower than that on a system (example 128 MB or 256 MB). The 4 GB of space is based of the 32-bit address space ($2^{32}$ bytes i.e. 4294967296 bytes). When any application executes the memory manager automatically maps the virtual address into physical addresses where the data really exists. Memory Management is the responsibility of the operating system, it allocates and de-allocates memory for applications (please do not think about malloc/new etc for now).

This 4 GB of virtual space is divided between the user mode and kernel mode equally. An application is typically loaded/execute in user mode memory, the kernel mode memory is where the kernel mode components are loaded/executed.

An application should not be able to directly access any kernel mode memory; any attempt to do so should result in an access violation. When an application needs to access / make calls to the kernel, a switch is made from user mode to kernel mode.

Thus the range of 0x00000000 – 0x7fffffff is for user mode and 0x80000000 – 0xBfffffff is for kernel mode, however you are allowed to change the allocated space with the /xGB switch in the boot.ini file, where x is the number of GB of memory for user mode.

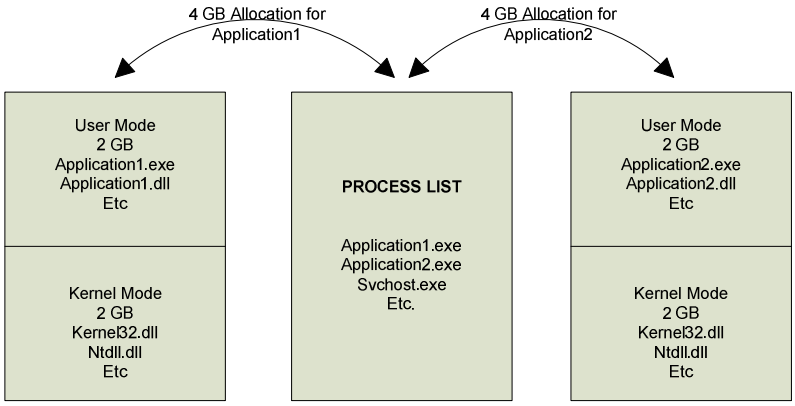


**Figure: Virtual allocation of space**

### DLL and EXE Files

Dynamic Linked Libraries (DLL), are binary files that contain subroutines linked together. These libraries (DLL's) are loaded by binary executables (.EXE) when they needs to use the subroutines built into the DLLs. DLL's and Exe files (executable binaries) are practically the same, they both use the same PE format except for a single bit that indicates to treat the binary as a DLL or EXE. Files with OCX (ActiveX) and CPL (Control Panel) extensions are exactly the same as DLLs as well.

Libraries can be static libraries or dynamic libraries. Windows mainly uses dynamic libraries which have a number of advantages including being loaded only once in memory and shared among multiple applications. Example of some DLLs are kernel32.dll, user32.dll etc.

## Memory Allocation

Each executable is loaded into unique non-overlapping address space. The memory location where a DLL for an application is loaded is exactly the same across multiple machines as long as the version of operating system and the application stays the same.

**Note:** While writing exploits, the knowledge of the location of a DLL and its corresponding functions will be used.

There are a number of tools available to view the address base where an executable is loaded. Microsoft provides a utility called dumpbin.exe with the default install of Visual Studio.

```
C:\>dumpbin /headers kernel32.dll
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights
reserved.
Dump of file kernel32.dll
. . . . . .
. . . . . .
        72000 base of data
        77E60000 image base (77E60000 to 77F45FFF)
```

Other tools such as Ollydbg (a tool which will become your best friend when you start doing any binary analysis or attempt to write exploits) and quickview plus can be used to view such information as well.

## Memory Overview

An application/process is loaded into three major memory areas, the stack segment, the data segment and the code/text segment.

The stack segment stores the local variables and procedure calls, the data segment stores static variables and dynamic variables and the text segment stores the program instructions.

The data and stack segment are private to each application, and no other application can access those areas. The text segment on the other hand is a read only segment which can be accessed by other processes too, however, if an attempt is made to write to this area, a segment violation occurs.

### *Memory Layout – Stack*

Stack is an area of reserved virtual memory used by applications. It is the operating system's method of allocating memory. A developer is not required to give any special instructions in code to augment memory, the operating system performs this task through guard pages automatically.

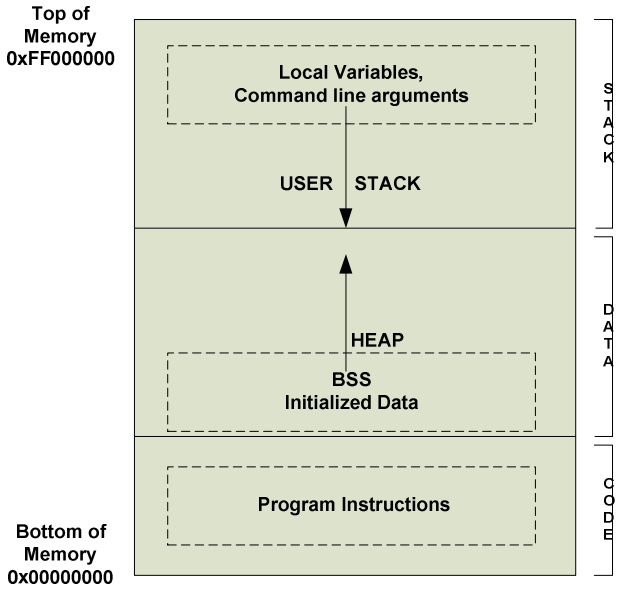


**Figure: Memory layout**

The following code would store the character array "var" on the stack.

Example:

```
char var[]="www.SecurityCompass.com";
```

The stack operates similar to a stack of plates in a cafe. The information is always pushed onto (added) and popped off (removed) from the top of the stack. The stack is a "Last In First Out" (LIFO) data structure.

Pushing an item onto a stack causes the current top of the stack to be decremented by four bytes before the item is placed on to the stack. When any information is added to the stack, all the previous data is moved downwards and the new data sits at the top of the stack. Multiple bytes of data can be popped or pushed onto the stack at any given time. Since the current top of the stack is decremented before pushing any item on top of the stack, the stack grows downwards in memory.

## Frame Layout

A stack frame is a data structure, which is created during the entry into a sub-routine/procedure (in terms of C/C++, creation of a function). The objective of the stack frame is to keep the parameters of the parent procedure as is and to pass arguments to the sub routine/ procedure. The current location of the stack pointer can be accessed at any given time by accessing the stack pointer register (ESP). The current base of a function can be accessed by using the EBP register which is called the base pointer or frame pointer and the current location of execution can be accessed by accessing the instruction pointer register (EIP).

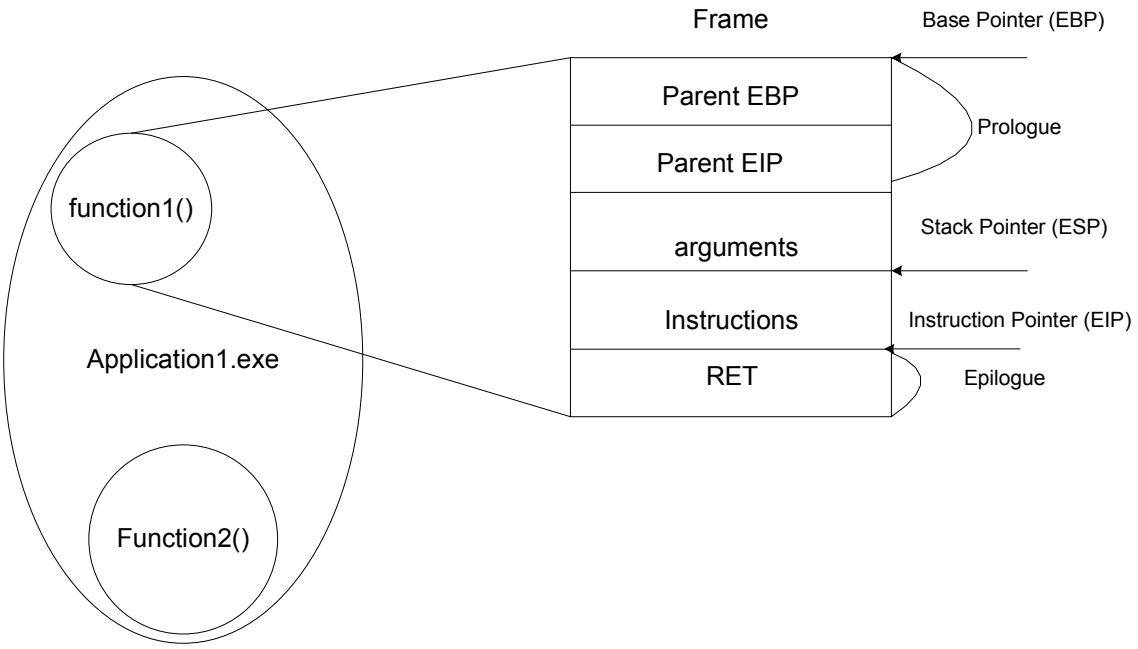


**Figure: Frame Layout**

### *Memory Layout – Heap*

Heap like stack is a region of virtual memory used by applications. Every application has a default heap space, however unlike stack, private heap space can be created by C/C++ programmers by using special instructions such as "new()" or "malloc()"and freed/cleared by using "delete()" or "free()". Heap operations are called when an application doesn't know the size of or the number of objects needed in advance or when an object is too large to fit onto the stack.

Example:

```
OBJECT *var = NULL;
var = malloc(sizeof (OBJECT));
```

The Windows Heap Manager operates above the Memory Manager and is responsible for providing functions which allocates or de-allocates chunks of memory. Every application starts out with a default of 1 MB (0x100000) of reserved heap size (view output from dumpbin below) and 4k (0x1000) committed if the image does not indicate the allocation size. Heap grows over time and it does not have to be contiguous in memory.

```
C:\WINDOWS\system32>dumpbin /headers kernel32.dll
<Deleted for brevity>
                100000 size of heap reserve (1 MB)
             1000 size of heap commit (4k)
<Deleted for brevity>
```

### Heap Structure

Each heap block starts of and maintains a data structure to keep track of the memory blocks that are free and the ones that are in use. Heap allocation has a minimum size of eight bytes, and an additional overhead of eight bytes (heap control block).

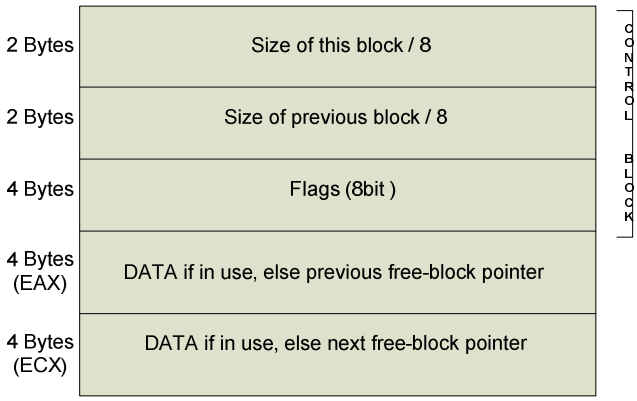


**Figure: Heap Layout**

The heap control block among other things also contains pointers to next free block. As and when the memory is freed or allocated, these pointers are updated.

***Note: Heap in XP SP2 and Windows 2003***

Microsoft has introduced an additional random value of 1 byte long which is stored in the control block of the heap. If this value is tampered with an error is generated, however, it is important to remember that since it is only a byte long, there are only 255 possibilities to guess this value (a possible brute forcer can be written for this part)

***Note: Stack Protection with the "/GS" Flag***

With Microsoft Visual Studio .NET a new compiler flag namely the "/GS" Flag has been introduced. This when set (set by default) sets up a canary value between a variables declaration on the stack and return address. Review the "Hello World" code in the next part which is compiled with this compiler option set.

## Glossary:

- **Stack**: Stack is memory that is statically allocated and is managed by the windows memory manger.
- **Heap**: Heap is memory that is dynamically allocated.
- **Stack Frame**: Stack frame is a data structure created during the entry into a function.
- **Physical Memory**: The amount of physical memory that is actually present in a computer.
- **Virtual Memory**: Virtual memory is a feature implemented in operating systems to extend the amount of memory available to programs. It allows operating systems to use more memory than the computer actually has physically present. In Microsoft Windows this is done using a pagefile.
- **Binaries**: When code is compiled to produce either .exe, .dll or other executable files, these files are known as binaries. Dll and .exe files both use the exact format, the only difference is a single bit that indicates if the file should be treated as an EXE or as a DLL.
- **DLL**: Dynamic Linked Libraries as opposed to static libraries do not copy data into a executable or library at compile time. These binaries with extensions such as .OCX (ActiveX controls) and .CPL (Control Panel applets files) are also DLLs but with different extensions.
- **PE Format**: Portable executable format is the format a typical windows executable is compiled into.

## Utilities:

- **Ollydbg**: ollydbg is a GUI based user mode debugging utility. This utility is very useful if you plan to get deep into writing exploits. This is available from http://www.ollydbg.de/.
- **Dumpbin**: Dumpbin is a utility that is installed with Visual C++. It is not in the default path of the users. It is typically installed in the "<visual studio home directly>\VC7\BIN\dumpbin".

## Links For Additional Reading:

- http://www.labri.fr/Perso/~betrema/winnt/ This site has links to articles on memory management.
- http://developer.intel.com/design/processor/ Intel's website has assembly language guides (Intel Software Developers Guide) with examples of assembly code and basic instruction they are among the best reference manuals for assembly for windows.